REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1992	Technical Paper

4. TITLE AND SUBTITLE
Advanced Techniques in Reliability Model Representation and Solution

5. FUNDING NUMBERS
WU 505-64-10-07

6. AUTHOR(S)
Daniel L. Palumbo and David M. Nicol

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17048

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TP-3242

11. SUPPLEMENTARY NOTES
Palumbo: Langley Research Center, Hampton, VA; Nicol: College of William and Mary, Williamsburg, VA.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 66

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The current tendency of flight control system designs is towards increased integration of applications and increased distribution of computational elements. The reliability analysis of such systems is difficult because subsystem interactions are increasingly interdependent. Researchers at NASA Langley Research Center have been working for several years to extend the capability of Markov modelling techniques to address these problems. This effort has been focused in the areas of increased model abstraction and increased computational capability. The reliability model generator (RMG) is a software tool that uses as input a graphical object-oriented block diagram of the system. RMG uses a failure modes-effects algorithm to produce the reliability model from the graphical description. The ASSURE software tool is a parallel processing program that uses the semi-Markov unreliability range evaluator (SURE) solution technique and the abstract semi-Markov specification interface to the SURE tool (ASSIST) modelling language. A failure modes-effects simulation is used by ASSURE. These tools were used to analyze a significant portion of a complex flight control system. The successful combination of the power of graphical representation, automated model generation, and parallel computation leads to the conclusion that distributed fault-tolerant system architectures can now be analyzed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Distributed fault-tolerant systems; Dynamically reconfiguring networks; Failure modes-effects analysis; Markov models; Reliability; State-space reduction	17
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

NASA-Langley, 1992

Nomenclature

AIPS	advanced information processing system
ASSIST	abstract semi-Markov specification interface to SURE tool
ASSURE	none
CH	channel
DIU	device interface unit
FMEA	failure modes-effects analysis
FMES	failure modes-effects simulation
FTP	fault-tolerant processor
HARP	Hybrid Automated Reliability Predictor
IAPSA	integrated airframe/propulsion control system architecture
I/O	input/output
NET	network
NI	network interface
RM	redundancy management
RMG	reliability model generator
SURE	semi-Markov unreliability range evaluator

Abstract

The current tendency of flight control system designs is towards increased integration of applications and increased distribution of computational elements. The reliability analysis of such systems is difficult because subsystem interactions are increasingly interdependent. Researchers at NASA Langley Research Center have been working for several years to extend the capability of Markov modelling techniques to address these problems. This effort has been focused in the areas of increased model abstraction and increased computational capability. The reliability model generator (RMG) is a software tool that uses as input a graphical object-oriented block diagram of the system. RMG uses a failure modes-effects algorithm to produce the reliability model from the graphical description. The ASSURE software tool is a parallel processing program that uses the semi-Markov unreliability range evaluator (SURE) solution technique and the abstract semi-Markov specification interface to the SURE tool (ASSIST) modelling language. A failure modes-effects simulation is used by ASSURE. These tools were used to analyze a significant portion of a complex flight control system. The successful combination of the power of graphical representation, automated model generation, and parallel computation leads to the conclusion that distributed fault-tolerant system architectures can now be analyzed.

Introduction

High reliability in digital systems is achieved, in a typical design, through redundancy and dynamic reconfiguration. Markov model solution techniques are commonly used when computing the reliability of this type of system. The state transition matrix representation of a Markov model is useful for expressing the sequence dependencies that can occur during a series of system failures and subsequent recoveries. However, distributed, fault-tolerant, and real-time systems result in extremely large and complex models. One conclusion of the integrated airframe/propulsion control system architecture (IAPSA) program (ref. 1) is that two factors limit the use of Markov models on the systems being proposed for the next generation of aerospace vehicles.

The first factor limiting the use of Markov models is that the state space grows exponentially with system size. This growth confines the size of the system that can be analyzed to one that can be accommodated by the available computing resources. One example is the Hybrid Automated Reliability Predictor (HARP) (ref. 2). The HARP program presents the user with a high-level interface consisting primarily of fault tree input (to describe system failure states) and fault/error-handling models (to describe recovery processes). This input is then translated into a Markov model and solved. To limit the size of the reliability model, HARP uses a process of behavioral decomposition, aggregation, and truncation at the third level. An estimate of the resulting model size for a system with n components is given by

$$\text{Total number of states} = \binom{n}{1} + \binom{n}{2} + \binom{n}{3} \tag{1}$$

Now, consider the IAPSA architecture, which consists of over 500 components. The approximation in equation (1) yields 21 million states. This approximation does not consider that, as in IAPSA, component dependencies limit the extent to which states can be aggregated. As discussed in a subsequent section, an IAPSA submodel with 80 components produced 27 million states. The magnitude of this problem is enormous.

The second factor limiting the use of Markov models is the difficulty in constructing a model of a large distributed and integrated system. The complex interdependencies confound the analyst's understanding of system behavior. Again, with IAPSA as an example, a single failure of a processing channel has the potential to effect three redundancy management regimes: the processor, the I/O network, and the I/O devices. These relationships, which can be significant, are at times obscured and threaten the accuracy of the model.

Researchers at NASA Langley Research Center have been working for several years to extend the

capability of Markov solution techniques to systems like IAPSA. These efforts have their foundation in the semi-Markov unreliability range evaluator (SURE) (refs. 3 and 4) and the abstract semi-Markov specification interface to the SURE tool (ASSIST) (refs. 5 and 6). The more recent efforts that are the subject of this paper include the reliability model generator (RMG) (refs. 7 and 8) and ASSURE. RMG is based on an algorithm for automating the failure modes-effects analysis (FMEA) that is part of every reliability analysis. RMG uses a graphically based object-oriented description of the system as input to this algorithm. The output of RMG is an ASSIST language description of the reliability model. ASSURE combines the ASSIST language with the SURE computational technique in a parallel program. ASSURE does not need to retain state information and therefore does not suffer from the state-space storage problem. ASSURE has also extended the ASSIST syntax to allow reference to a failure modes-effects simulation (FMES). Features such as graphical representation, automated model generation, parallel processing, and FMES are being combined into a tool set that will presumably have the power to compute the reliability of large fault-tolerant flight control systems.

In the following sections, three submodels of the IAPSA architecture are introduced as a basis for discussing RMG, ASSURE, and FMES. The next section is a brief description of IAPSA.

IAPSA Architecture

The integrated airframe/propulsion control system architecture (IAPSA) (ref. 1) was designed to meet the requirements generated when airframe and engine control laws are combined in a high-performance military aircraft. Features of the aircraft are canards and dual engines with variable inlets and vectoring nozzles.

Figure 1 is a representative block diagram of the IAPSA architecture. The architecture is based on the advanced information processing system (AIPS) building block elements (ref. 9). The AIPS building blocks have been designed to provide fundamental system resources for a wide spectrum of aerospace applications. The building blocks include fault-tolerant processors (FTP's), network interfaces (NI's), nodes, links, and device interface units (DIU's). The FTP's can be configured as quad or triplex redundant computers. Nodes and links are used to construct repairable mesh networks. In operation a mesh network is configured as a bus; that is, the links on each node are statically enabled or disabled such that every node can be reached. The I/O devices are connected to the network through the DIU's. If a failure occurs on the network, the path with the failure is disabled and an alternate path is enabled. If this repair can be accomplished quickly, one mesh network can service the entire vehicle. In practice, using two networks is necessary, one to control the aircraft while the other is repaired.

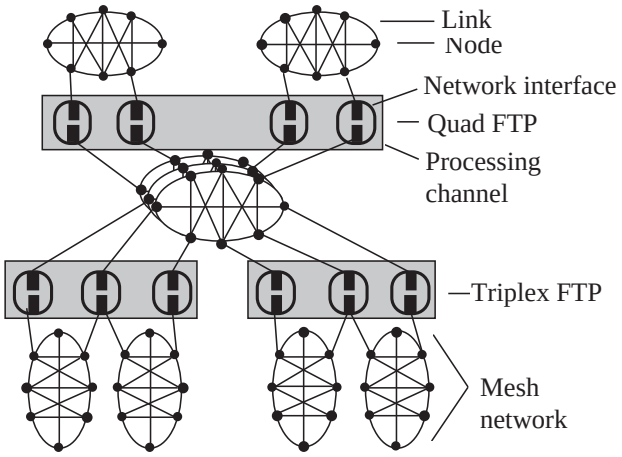

Figure 1. IAPSA architecture.

A quad FTP has the major responsibility for airframe control. Connected to it are two I/O mesh networks, one of which must be functioning for safe operation of the aircraft. A triplex FTP is used for each engine where again dual mesh networks handle the I/O traffic. A triplex mesh network provides a highly reliable data path for interprocessor communication. In total, the architecture consists of 10 processor channels, 20 NI's, 50 nodes, 90 links, 36 DIU's, and 300 I/O devices for a total of 490 components. This design does not include the components necessary to establish the interprocessor link. This part of the system has not yet been designed, but analysts estimate that about 100 NI's, nodes, and links would be used to implement it.

Reliability Model Generator

The reliability model generator was designed as a tool for system designers. Working from a data base of building blocks, designers can construct a graphical block diagram of the system. When the design is finished, an automatic failure modes-effects analysis is performed with data associated with the graphical building block objects. The result of the FMEA is then translated into a reliability model in the ASSIST language (refs. 5 and 6).

The automated FMEA is implemented with an object-oriented data base approach conceived by The Boeing Company (refs. 7 and 8). In the data base, a

building block has graphical attributes of the building block itself as well as its inputs and outputs. The building block has data attributes of component modes and mode transition functions. The input and output have data attributes of effect messages and output transition functions.

Examples of component modes are GOOD, FAILED ACTIVE, and FAILED PASSIVE. Component modes are closely related to reliability model state variables. Mode transition functions control mode state changes. The mode transition functions are similar to the transition rules found in ASSIST. A mode transition function can have as its input the current mode, the value of the building block input and output, and a rate. Thus, a mode transition function can specify that if a building block is GOOD, then it may become FAILED ACTIVE at rate λ.

Building block output effect messages take on values such as NOMINAL, ERROR, and NONE. Output transition functions control the value of the messages. Output transition functions have as their input the building block input and current component mode. Output transition functions are considered to be an instantaneous evaluation of building block behavior. These functions are loosely related to the death conditions found in ASSIST. For example, an output transition function can specify that an output effect is ERROR if either the component mode is FAILED ACTIVE or an input effect is ERROR.

To perform the automated FMEA, a building block representing the system is formed with an output that reflects the system's condition and with inputs from other building blocks. RMG is then directed to analyze the system for conditions leading to an ERROR output of the system. A backward-chaining technique is used to trace this failed state throughout the system. As the state is traced, this technique constructs the core of the reliability model.

Example 1: FTP Network Interface

Figure 2 is a diagram of the first example, which focuses on the interaction of the FTP with the mesh networks. Here the mesh networks are modelled as single, repairable components. Three FTP channels are connected to each network so that both networks function in the event of two channel failures. Initially FTP channel 1 (CH1) is controlling network 1 (NET1) with network interface 1 (NI1) and CH4 is controlling NET2 with NI6. The remaining connections are disabled. While appearing simple on the surface, this model is rich in interdependencies.

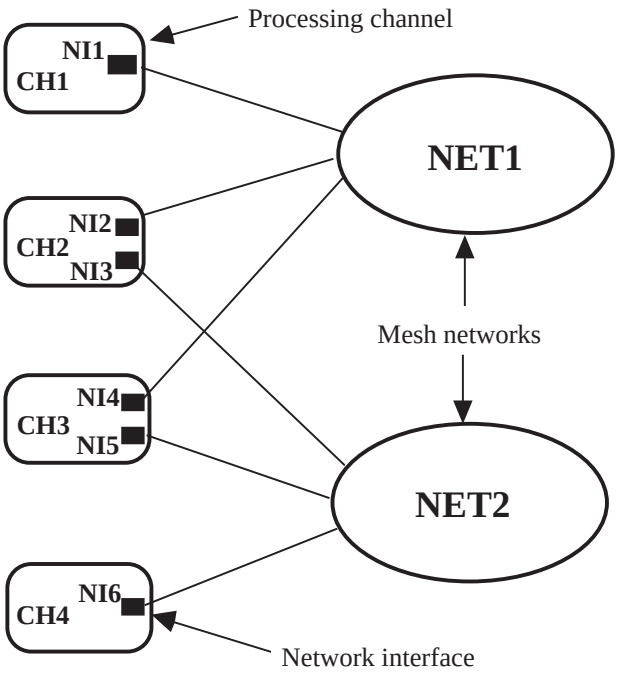

Figure 2. Example 1: FTP network interface.

Because CH1 initially controls NET1 and CH4 controls NET2; four network interfaces (NI2, NI3, NI4, and NI5) are not used at this time. An FTP channel failure causes the failure of its NI unit(s). Thus, a failure of CH1 causes NI1 to fail. The effect of this failure depends on whether or not that particular NI unit was controlling its associated network at the time of the failure. For example, if CH1 fails from initial conditions (i.e., NI1 is controlling NET1), then two recovery mechanisms must be activated: one to repair the FTP by disabling CH1 and the other to repair NET1 by enabling NI2 as controller of NET1. If NI1 fails from the initial state, then a network recovery disables the failed link and enables NI2 as the NET1 controller on the condition that CH2 and NI2 have not yet failed. A subsequent failure of CH1 results only in a recovery of the FTP because CH1 is not the current network controller. A reliability analysis tool must be able to track such dependencies without burdening the user with cumbersome constructs or cryptic tricks. (See ref. 10 for further discussion.)

Figure 3 is the RMG block diagram for example 1. This model contains all the elements of figure 2 with the addition of building blocks representing the redundancy-management (RM) routines (FTP RM and NETn RM) and the SYSTEM building block. The following description lists the component attributes and explains how RMG uses these attributes to perform the automated FMEA.

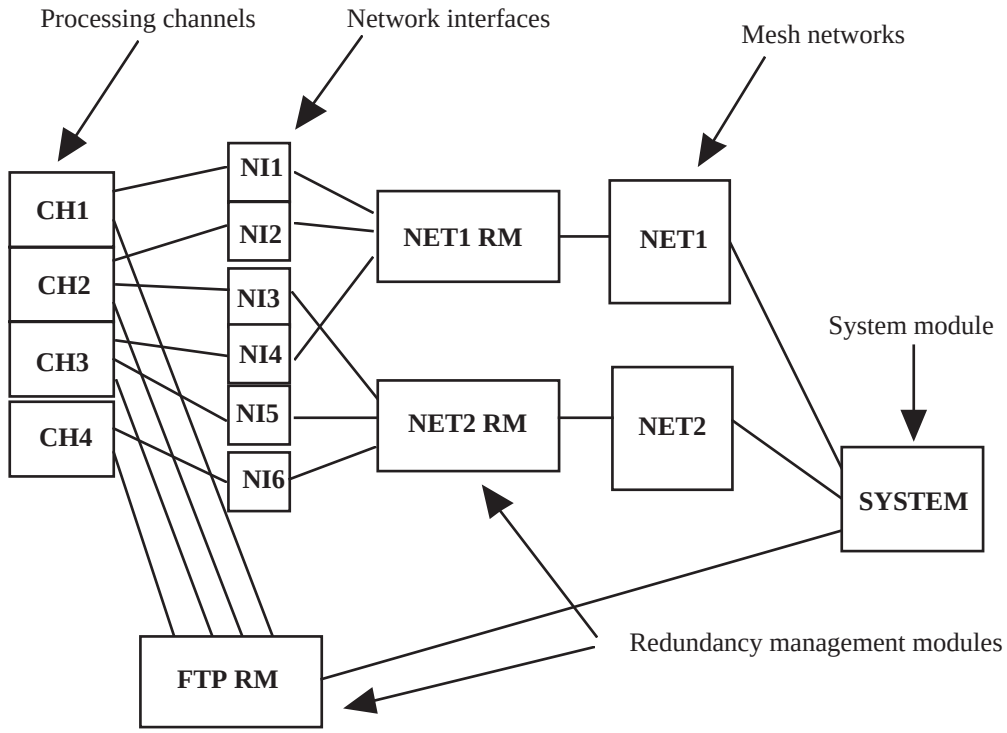

Figure 3. RMG diagram for example 1.

The FTP channels (CH1–4) have the following attributes:

Component modes:	(GOOD, FAILED, REMOVED);
Inputs:	;
Outputs:	CH_STATUS;
Mode transition functions:	IF (mode=GOOD) THEN (mode=FAILED) AT failure_rate; IF (mode=FAILED) THEN (mode=REMOVED) AT recover_rate;
Output modes:	(NOMINAL, ERROR, NONE);
Output transition functions:	IF (mode=GOOD) THEN (CH_STATUS=NOMINAL) ELSE IF (mode=FAILED) THEN (CH_STATUS=ERROR) ELSE IF (mode=REMOVED) THEN (CH_STATUS=NONE);

The FTP RM has the following attributes:

Component modes:	();
Inputs:	CH_STATUS_1,CH_STATUS_2,CH_STATUS_3, CH_STATUS_4;
Outputs:	FTP_STATUS;
Mode transition functions:	;
Output modes:	(NOMINAL, ERROR);
Output transition functions:	IF number_of ((CH_STATUS_1=NOMINAL), (CH_STATUS_2=NOMINAL), (CH_STATUS_3=NOMINAL), (CH_STATUS_4=NOMINAL)) > number_of ((CH_STATUS_1=ERROR), (CH_STATUS_2=ERROR), (CH_STATUS_3=ERROR), (CH_STATUS_4=ERROR)) THEN FTP_STATUS=NOMINAL ELSE FTP_STATUS=ERROR;

The FTP RM block is an instantaneous evaluation of the state of the FTP and thus does not require component modes or mode transition functions. The RMG provides a convenient number_of function that accumulates the number of TRUE conditions found in the argument list. Here, the output transition function uses the number_of function to perform a simple majority vote evaluation. In the case of a quad vote, two or more inputs receiving ERROR status cause the FTP RM block to transmit an ERROR status. The effect of a recovery of a failed channel is to send a NONE status, which protects the FTP from failure on a subsequent channel failure.

The NI components have the following attributes:

Component modes:	(GOOD, FAILED);
Inputs:	(CH_STATUS);
Outputs:	(NI_STATUS);
Mode transition functions:	IF (mode=GOOD) THEN (mode=FAILED) AT failure_rate;
Output modes:	(NOMINAL, ERROR);

Output transition functions:	IF (mode=GOOD) and (CH_STATUS=NOMINAL) THEN
	NI_STATUS=NOMINAL
	ELSE
	NI_STATUS=ERROR;

The formulation of the output transition function causes the NI component to produce an error message output when the host channel fails. Thus, to those components connected to the NI outputs, the NI itself appears to have failed.

The NET RM components have the following attributes:

Component modes:	(MODE1, MODE2, MODE3);
Inputs:	NI_STATUS_1, NI_STATUS_2, NI_STATUS_3;
Outputs:	NET_RM_STATUS;

Mode transition functions:	IF (mode=MODE1) and (NI_STATUS_1=ERROR) THEN
	IF (NI_STATUS_2=NOMINAL) THEN
	(mode=MODE2) AT recovery_rate;
	ELSE IF (NI_STATUS_3=NOMINAL) THEN
	(mode=MODE3) AT recovery_rate;
	IF (mode=MODE2) and (NI_STATUS_2=ERROR) THEN
	IF (NI_STATUS_3=NOMINAL) THEN
	(mode=MODE3) AT recovery_rate;

Output modes:	(NOMINAL, ERROR);

Output transition functions:	IF (mode=MODE1) THEN
	(NET_RM_STATUS=NI_STATUS_1);
	IF (mode=MODE2) THEN
	(NET_RM_STATUS=NI_STATUS_2);
	IF (mode=MODE3) THEN
	(NET_RM_STATUS=NI_STATUS_3);

The NET RM block uses the status outputs of the three NI components to determine its operating mode. The operating mode corresponds to which NI (and therefore which FTP channel) is controlling the network. The status of the controlling NI is propagated as the NET RM output to the NET component.

The NET components have the following attributes:

Component modes:	(GOOD,FAILED);
Inputs:	NET_RM_STATUS;
Outputs:	NET_STATUS;
Mode transition functions:	IF (mode=GOOD) THEN (mode=FAILED) AT failure_rate; IF (mode=FAILED) THEN (mode=GOOD) AT recovery_rate;
Output modes:	(NOMINAL, ERROR);
Output transition functions:	IF (mode=GOOD) THEN (NET_STATUS=NET_RM_STATUS) ELSE (NET_STATUS=ERROR);

The NET component is assumed to be infinitely repairable; that is, the NET has inexhaustible spares. However, when the NET RM indicates an NI failure, the NET propagates an ERROR indication until the NET RM replaces the failed NI (if possible).

The SYSTEM building block has the following attributes:

```
Component modes:            ();
Inputs:                     NET_STATUS_1, NET_STATUS_2, FTP_STATUS;
Outputs:                    SYSTEM_STATUS;
Mode transition functions:  ;
Output modes:               (NOMINAL,ERROR);
Output transition functions: IF ( (NET_STATUS_1=NOMINAL) or
                                  (NET_STATUS_2=NOMINAL) ) and
                                  (FTP_STATUS=NOMINAL) THEN
                                    (SYSTEM_STATUS=NOMINAL)
                             ELSE
                                    (SYSTEM_STATUS=ERROR);
```

The SYSTEM building block contains an output transition function that modifies the SYSTEM mode from NOMINAL to ERROR in the event that either both network output effect messages are ERROR or the FTP RM output message is ERROR.

The SYSTEM building block is used as a starting point for the FMEA. The conditions in the SYSTEM output transition that contribute to an ERROR condition are traced back, assembled, and reduced to disjunctive normal form.[1] These conditions can then be listed as DEATHIF statements in the ASSIST model description. Mode transition functions are resolved and used as model expansion rules (called TRANTO rules) in ASSIST.

Results

Models of example 1 were both manually coded and automatically generated with RMG. The models appeared to be very different. RMG produced an exhaustive expansion of the system. The manually coded model was more compact in a situation analogous to comparing manually written assembler code to compiler-generated code. The computed reliability for the two models differed by a small amount (fifth decimal digit). Comparing the models for equivalence uncovered an interesting discrepancy. The RMG-generated model achieved a more thorough expansion of the state space. In the manually coded model, some network failures were inadvertently omitted. The RMG-generated model took about five times as long to process because of both the size of the model representation (the ASSIST code) and the larger state space that RMG covered. The automatically generated model size is almost three times larger than the manually coded model (see table I).

Table I. Example 1 Performance Metrics

Parameter	Model	
	Manually coded	RMG generated
Number of states	1555	4466
ASSIST time	105 sec	1810 sec
SURE time	420 sec	633 sec

Discussion

The particular diagram shown in figure 3 is not an ideal graphical representation. Having the display of subcomponents (such as the NI's) somehow represent the particular relationship between the subcomponents and their parent components is preferred. For example, the NI communicates with an FTP channel and is critically dependent on the FTP channel. The NI is a subcomponent and should be viewed as such. (See fig. 2.)

Redundancy management routines are more difficult to represent. A redundancy management routine is what turns a discrete set of computer channels into a fault-tolerant computer. Yet, the redundancy management routine is not a component typically pictured in a block diagram as it is in figure 3. As for the FTP, the FTP RM might be better expressed by explicitly showing the interchannel linkages and voters as subcomponents that are part of the actual FTP channel architecture. However, this type of expres-

[1] Given a logical expression that consists of a series of subexpressions that are connected by AND or OR, disjunctive normal form is a reduction of the logical expression to one that is a series of subexpressions connected by OR where the subexpressions contain only AND logical functions.

sion does not work for the NET RM. The NET RM organizes the NI's, which are part of an FTP channel, and the mesh network into a fault-tolerant network. As modelled in this example, network recoveries are generated in both the NET RM and the NET component. This adaptation was unavoidable because of limitations in the version of RMG used to generate this example. When considering a better way to represent the NET RM, it is difficult to imagine a clean construct that can be added to each component of this assemblage and be able to describe the NET RM function. The redundancy management routines are thus best described as separate objects whose attributes can be related to other components with a graphical device such as color or a unique icon. This concept will be considered in future versions of the software.

ASSURE

Given the capability to automatically generate a model, the problem immediately becomes one of computing the extremely large models that will certainly follow. The ASSIST/SURE combination has the drawback that the entire state space must be generated by ASSIST and searched by SURE. While methods of *pruning* the state space and path depth have been developed for both ASSIST and SURE, modest models of a few dozen interdependent components quickly tax current workstations.

The ASSIST modelling language has been combined with the SURE solution technique in a reliability analysis tool (ASSURE) in which state-space storage requirements are minimized. The SURE solution technique provides for the calculation of a Markov model as the expansion of a series of independent paths (ref. 4). The ASSIST modelling language describes how these paths are grown (ref. 5). In ASSURE, the ASSIST language is translated into C, linked with SURE solution procedures, and executed to solve the model. The state probabilities can then be calculated as the model is grown. Two mechanisms are available to reduce model size. With access to the state probabilities, an informed decision can be made as to when to terminate path growth (e.g., when state probability $<10^{-14}$). Also, because the only state of consequence at any time is the state being expanded, when expansion is complete, the state can be discarded. Thus, ASSURE does not need to maintain the complete state space in memory. Also, because the paths through the model are independent, the ASSURE program can be parallelized.

Example 2: Nodes, Links, and Devices

Figure 4 illustrates a problem generated to test the capability of ASSURE. The system is an evolution of example 1 with the addition of a two-layer network and I/O devices. A mesh network could not be modelled initially because of the difficulty in expressing the network regrow algorithm in the ASSIST language. (This difficulty was later rectified. See section entitled "Failure Modes-Effects Simulation.") The I/O devices are quad redundant, use majority voting, and have redundancy management routines similar to those of FTP.

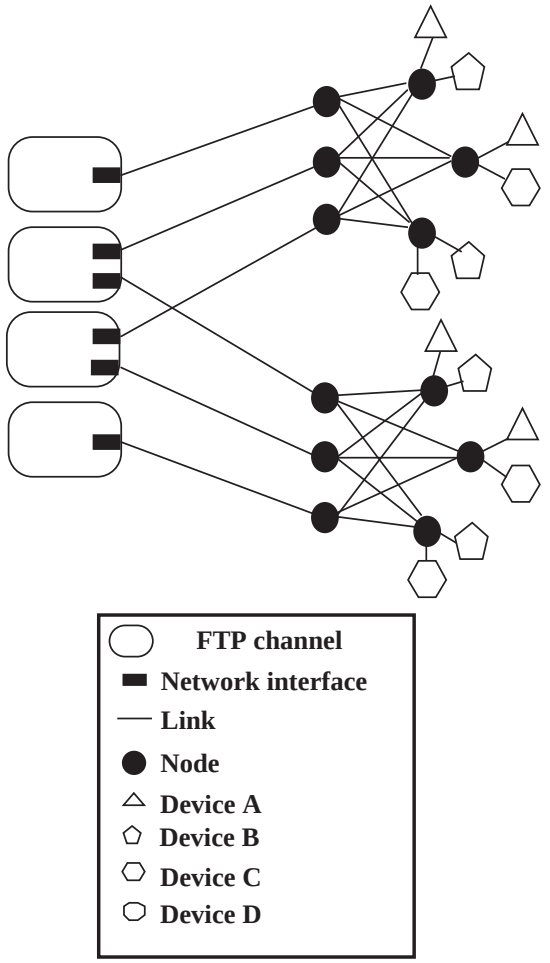

Figure 4. Example 2: FTP network interface with two-layer network and I/O devices.

Results

ASSIST produced a reliability model for the system in figure 4; the model contained over 40 000 states and 1 000 000 transitions (with no pruning). Direct comparison with ASSURE is not possible because ASSURE does not aggregate states when it produces the model. Model statistics (reliability and pruning bounds, number of pruned paths,

Table II. Example 2 Performance Metrics

	Processor	SURE model size	Run time, hr	Memory usage
ASSIST/SURE	SUN 3/150	27 Mbyte	11.50	100 Mbyte
ASSURE (Serial)	SUN 3/150	NA	0.60	1 Mbyte
ASSURE (Parallel)	32 iPSC/860	NA	0.01	1 Mbyte per node

and pruning error) for SURE and ASSURE were identical; thus, ASSURE computed the model correctly. ASSURE exists both in serial and parallel form. The test runs for the serial version of ASSURE and ASSIST/SURE were performed on a SUN 3/150 processor. The parallel version of ASSURE was executed on a 32-node iPSC/860 hypercube. The serial ASSURE program execution was 10 times faster and used 100 times less memory than ASSIST/SURE. Parallel ASSURE increased this performance another 100 times. (See table II for details.) Overall, a speed increase on the order of 3 orders of magnitude is realized over the original ASSIST/SURE solvers. The processors in the hypercube are typically over 90 percent utilized.

Note, ASSURE is a prototype and thus does not perform extensive error checking (as does SURE). If extensive error checking were performed it would reduce the observed improvement. However, given the degree of efficiency of the parallel version, a great deal of improvement will always be obtained with parallelization. Serial ASSURE benefits from not having to maintain the complete state space in memory while computing. As soon as the state space outgrows available physical memory, ASSIST/SURE suffers performance degradation due to swapping of virtual memory.

Failure Modes-Effects Simulation

As previously mentioned, expressing the mesh network regrow algorithm in the ASSIST language is difficult. Two possible methods are exhaustive enumeration (which is almost immediately ruled out) and the division of the algorithm into discrete steps. The division method is possible but presents a confusing model because each step in the process must be assigned a rate and therefore produces another state with subsequent children states.

An alternative approach takes advantage of the ASSURE translation of ASSIST into C code. Thus, the regrow algorithm can be coded in a C procedure and ASSURE can reference this procedure at the appropriate time. Studying this approach revealed that an extension of the ASSIST syntax was necessary. Further work using the extension to ASSIST led to an approach in which the concept of automated FMEA fostered by RMG is incorporated as C code in ASSURE. This concept is the failure modes-effects simulation.

ASSIST Extensions

The basic components of an ASSIST model description are the state vector, model expansion rules, and model termination rules. The reliability model is produced by repeatedly applying the model expansion rules to a state vector and thus creating new state vectors. The process continues until the list of state vectors is exhausted. A model expansion rule (called a TRANTO statement) is composed of a conditional expression, a state translation expression, and a rate. A transition in a reliability model is thus completely defined by its starting state (identified by the conditional expression), its ending state (defined by the translation expression), and the rate at which the transition occurs. Model growth is terminated by checking the new state against the model termination rules (conditional expressions called DEATHIF statements). Death states are not expanded. System unreliability is calculated as the total probability of entering a death state before the end of the mission time.

The ASSIST language was extended to allow reference to two types of C functions termed *conditional functions* and *effect functions*. A conditional function takes as input the state vector and returns a value of TRUE or FALSE. Conditional functions are used in DEATHIF statements and the conditional part of TRANTO statements. An effect function is used in place of the state translation expression of a TRANTO statement.

Figures 5(a) and 5(b) show models of a simple quad FTP in standard ASSIST (fig. 5(a)) and extended ASSIST (fig. 5(b)). In standard ASSIST, the model begins with the declaration of two transition rate constants. This declaration is followed by a SPACE statement that defines the state vector as two four-element arrays (CH_G and CH_B). These arrays are of type Boolean and indicate whether FTP channels are GOOD (CH_G) or BAD (CH_B). In the

START statement, the state vector is initialized to all channels being GOOD. The DEATHIF statement supplies a majority vote termination condition. Finally, two TRANTO statements (IF ... TRANTO ... BY ... ;) supply model growth rules. The TRANTO statements are embedded in a FOR loop to scan each element of the state vector arrays.

```
CH_Fail_Rate = 1.0E-4;
FTP_Recovery = 3.0E4;
SPACE = (CH_G: array[1..4], CH_B: array[1..4]);
START = (4 of 1, 4 of 0);
DEATHIF CH_B[1]+CH_B[2]+CH_B[3]+CH_B[4] >=
        CH_G[1]+CH_G[2]+CH_G[3]+CH_G[4];
FOR i=1,4
 IF CH_G[i]=1 TRANTO
    CH_G[i]=0, CH_B[i]=1 BY CH_Fail_Rate;
 IF CH_B[i]=1 TRANTO
    CH_B[i]=0 BY FTP_Recovery;
ENDFOR;
```

(a) FTP model in standard ASSIST.

```
CH_Fail_Rate = 1.0E-4;
FTP_Recovery = 3.0E4;
SPACE = (FTP, CH: array[1..4]);
START = (5, 4 of 5);
DEATHIF ERRFTP();
FOR i=1,4
  IF GOOD(CH[i]) TRANTO
     CH_FailEff(i) BY CH_Fail_Rate;
 IF RECOVER(FTP) TRANTO
    FTP_RecEff()  BY FTP_Recovery;
ENDFOR;
```

(b) FTP model in extended ASSIST.

Figure 5. Simple quad FTP models.

In the extended ASSIST model (fig. 5(b)), notice the conditional function calls ERRFTP(), GOOD(), and RECOVER() and effect function calls CH_FailEff() and FTP_RecEff(). The model in figure 5(b) also reflects a different modelling strategy, which is a natural result of the FMES process. Consider the state vector. Two entities are modelled in this system: actual physical components called channels (CH[i]) and a super component called FTP. The FTP is a logical entity whose state is a collective function of the channels' states. Also, these components no longer have simple Boolean values but can take on a range of values as follows:

GOOD = 1;
ACTIVE = 2;
IN_USE = 4;
ERROR = 8;
RECOVERING = 16;
ELIMINATED = 32;

These values represent single bits in the state variable and can be combined to define a component's state. Thus, a component can be GOOD (with state = 1) or a component can be GOOD and IN_USE (with state = 5). The GOOD + IN_USE value is used to initialize the state variables in the START statement in figure 5(b). Defining macros to operate on the state variables is often helpful. The following macros are used in the FMES code:

SetRecovery(v):	Sets the RECOVERING bit.
SetFailError(v):	Sets the ERROR bit and clears the GOOD bit.
SetElim(v):	Sets the ELIMINATED bit.
SetNotInUse(v):	Clears the IN_USE bit.
GoodInUse(v):	Tests state variable for both GOOD and IN_USE bits.
ErrorInUse(v):	Tests state variable for both ERROR and IN_USE bits.

A Simple FMES

Figures 5(a) and 5(b) are practically identical with the exception that functions written in standard ASSIST have been replaced by function calls in extended ASSIST. Conditional functions can take as parameters one or more state variables. Effect functions can pass an integer argument for array indexing. The primary benefit of using extended ASSIST is that complex state transitions such as a network repair can be coded in algorithmic form instead of the exhaustive enumeration sometimes necessary with standard ASSIST. A secondary benefit is that the resulting ASSIST model is less complicated and thus more readable.

The ASSIST TRANTO statement contains three expressions: a condition, a destination state translation, and a rate. The failure modes-effects simulation describes the destination state translation as a chain reaction among the components of the system using the concepts of component modes and mode effect messages developed in RMG. The FMES functions are grouped into two categories: effect functions (which are referenced in ASSIST TRANTO state translation expressions) and dependency functions. An effect function links the FMES with the ASSIST model. The dependency functions propagate the effect throughout the system while making the appropriate state changes. Figure 6 shows the FMES for the model of figure 5(b).

According to the first TRANTO in figure 5(b), if a CH is GOOD, then it can fail with effect determined by CH_FailEff(). The function CH_FailEff first modifies the channel's state to FAIL + ERROR, then it calls dependency function FTP_Dependson_CH(). (See fig. 6.) The FTP dependency function uses the voter majority rule to determine the state of the FTP. FTP should recover if any channel is producing errors, and FTP is failed if the error-producing channels outnumber the good channels. Setting FTP to a recovering state enables the second transition, in figure 5(b); this transition uses FTP_RecEff() to obtain the effect of the FTP recovery. In FTP_RecEff(), error-producing channels are set to not in use and eliminated from the system.

```
CH_FailEff(my_id)
 int my_id;
 {
    SetFailError(CH[my_id]);
    FTP_DEPENDSON_CH();
 }
 FTP_DEPENDSON_CH()
 {
    int i,g,b;
    g=0;  b=0;
    for (i=1;  i<=4; i++)
      {
        if (GoodInuse(CH[i]) && !Error(CH[i])) {g++;}
        else if (ErrorInuse(CH[i])) {b++;}
      }

 FTP = GOOD||INUSE;
 if (b!=0) (SetRecover(FTP);}
 if (b>=g) || (g==0) {SetFailError(FTP);}
}
FTP_RecEff()
 {
    int i;
    for (i=1; i<=4; i++)
      if (ErrorInuse(CH[i]))
        {
          SetNotInuse(CH[i]);
          SetElim(CH[i]);
         }

    FTP_DEPENDSON_CH();
}
```

Figure 6. FMES C code for figure 5(b).

Modelling With FMES

In the simple quad system, two types of transitions are modelled: failure transitions and recovery transitions. (However, others are possible.) Failure transitions can occur at any time to any component. The effect of the failure on the system state is determined by that component's fail effect function. Recovery transitions are most often enabled by a component's fail effect function (although they can be triggered by other effects). A recovery is brought about by a super component. A super component is a set of components that have been grouped together to increase reliability. The quad fault-tolerant computer is an example of a super component.

Super components are responsible for redundancy management. When a component fails, its fail effect function sets the RECOVER mode descriptor of that component's super component. A good example is the quad redundant fault-tolerant computer. This super component is called FTP and is composed of four channels. When a channel fails, its fail effect function sets the RECOVER flag in FTP. The FTP recovery effect functions are then called during the calculation of the now enabled recovery transition.

Super components do not have failure transitions, yet they are able to fail. Again, with the FTP as an example, majority voting is used among its set of channels to mask and detect errors. Whether or not the FTP super component is operating properly is a function of the state of the set of CH's assigned to the FTP as calculated in function FTP_DEPENDSON_CH(). A function sensing the state of FTP is constructed and called as a death condition. If the death condition is met, the FTP has failed and thus the system (in this case) has failed.

A brief description of how the FMES is used in ASSURE is as follows:

1. A set of mode descriptors and effect messages is defined and a state vector constructed.
2. Fail effect and recovery effect functions are defined.
3. Condition functions for failure and recovery transitions are defined.
4. Death condition functions are defined.
5. The model is executed for each component as follows:
   a. IF FAIL_CONDITION() TRANTO FAIL_EFFECT() BY RATE.
   b. IF RECOVERY_CONDITION() TRANTO RECOVERY_EFFECT() BY RATE.
   c. Test for DEATH_CONDITION() for each new state.
   d. Compute reliability as model is expanded, pruning where possible.
6. Print results.

Example 3: Mesh Network

Figure 7 illustrates the system configuration for this mesh network example. Two network partitions consisting of 7 nodes and 14 links interface FTP with 4 quad redundant I/O groups. The mesh network uses a regrow algorithm to repair failures. The I/O devices connect to the network through device interface units. This system contains over 80 components and 7 different redundancy management groups or super components (FTP, NET1, NET2, and four I/O devices). The computer, two networks, and four I/O devices are reconfigurable and controlled by the seven separate redundancy management routines. The recovery of a simultaneous failure of a channel of the fault-tolerant computer and a node of one of the networks requires two separate operations. However, the recovery from a simultaneous link and node failure on the same network can be accomplished in one operation.

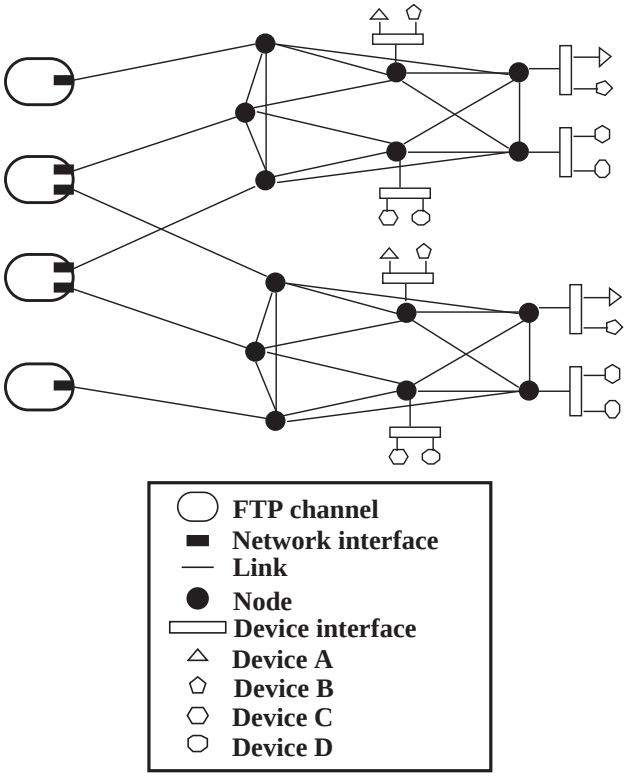

Figure 7. Example 3: Mesh network.

Component mode descriptors and mode message. As previously mentioned, the FMES is derived from the automated FMEA as used by RMG; thus, a set of mode descriptors and mode messages must be defined. Although different mode descriptors and messages can be defined for each component, it is best to seek, if possible, a set of common descriptors and messages that can be used throughout the system. The system in figure 7 is used as an example because it is large and complex and the set of descriptors and messages needed to define that system should suffice for most others.

Mode descriptors are implemented as bit values that have a meaning associated with their TRUE (set) and FALSE (reset) values. In the following descriptions, the first value is associated with TRUE and the value in parentheses is associated with FALSE.

GOOD (FAILED):	Describes the component's physical state. If the component is GOOD, it can FAIL at any time.
ACTIVE (PASSIVE):	Describes the nature of failure. An ACTIVE failure is able to produce erroneous behavior. A PASSIVE failure is analogous to failing safe.
ERROR (BENIGN):	States that detectable errors are being produced.
IN_USE (NOT_IN_USE):	Used primarily for modelling spares. For example, a component (such as a link) that is NOT_IN_USE might not affect the system with an active failure. Super component recovery effect functions control the value of this descriptor.
RECOVERING (NORMAL):	Used with super components to enable recovery transitions.
ELIMINATED (MEMBER):	Used in recovery effect functions to mark a component as having been removed from the set of good components.

A component typically begins in a GOOD + IN_USE state. A failure can cause it to transition to FAILED + IN_USE + ERROR and can cause its super component to transition from GOOD + IN_USE + NORMAL to GOOD + IN_USE + RECOVERING.

Mode messages have mutually exclusive values. Because the mode messages are not part of the state variable, they can have similar names to convey similar meaning.

NOMINAL:	Indicates that the sending component's current operation is within specification.
FAIL:	Indicates that the sending component has failed.
ERROR:	Indicates detectable erroneous behavior.
NONE:	Indicates passive failure.
NIU (Not In Use):	Indicates that the sending component has been switched to standby (as a spare).

Fail effect functions. A fail effect function is named by attaching the term "_FailEff" to a component's state variable name. For example, component CH has fail effect function CH_FailEff. A fail effect function has three stages. The first stage alters the component's mode, which can be, for example, from GOOD + IN_USE to FAILED + IN_USE + ERROR. A second function can then be called to send the appropriate effect messages to this component's neighbors. This function is named by attaching "_Dependents" to the component name (e.g., in CH_Dependents). Finally, a component calls zero, one, or more super component dependency functions. The super component dependency functions can be contained in the "_Dependents" function, but it is best to separate them because the super components are different from normal components.

Dependency functions. A primary dependency function interprets a component's mode and sends a message reflecting the component's new state to those other components that are immediately affected. The messages are sent through use of secondary dependency function calls of the form "X_Dependson_Y(X_id, Y_id, Y_message)," where Y is the local component. Thus, the function call X_Dependson_Y(X_id, Y_id, Y_message) is found in function Y_Dependents.

For example, consider the NI which resides in a channel of the FTP (CH). As a result of executing a failure transition for component CH[2], fail effect function CH_FailEff(2) is called. This function then calls the primary dependency function CH_Dependents(2). Because two NI's reside in CH[2], two secondary dependency function calls are made as follows:

```
NI_Dependson_CH(2,2,FAIL);
NI_Dependson_CH(3,2,FAIL);
```

The secondary dependency function alters the receiving component's state and then calls that component's primary dependency function. The effect of the failure is thus propagated throughout the system.

A super component dependency function (e.g., FTP_Dependson_CH) differs substantially from a normal component's dependency function. This difference occurs because a super component must have access to the state of all components in its domain. For example, FTP_Dependson_CH must be able to read the state of each of its channels to determine whether the voter function is error free. Also, in the case of the network, a single failed node has the effect of taking the network off-line until the network repairs. Thus, the super component function NET_Dependson_NODE must be able to alter the state of all nodes and links in the network to set them to NOT_IN_USE.

Recovery effect functions. A recovery effect function is named by attaching the term "_RecEff" to the super component's name (e.g., FTP_RecEff). A recovery effect function examines and alters, if necessary, the state of each of the components in its domain. For example, after failing, a CH is in mode FAILED + IN_USE + ERROR. The recovery function changes this to FAILED + NOT_IN_USE + ERROR + ELIMINATED; the device is now no longer in use or part of the spare pool. The recovery function then calls the component's primary dependency function to propagate the effect of the mode changes.

Effect of CH[1] failure. The complete FMES for this system is not given here because of the amount of detail. The following description explains what happens when CH[1] fails:

CH_FAILEFF():	CH[1] set to FAILED + IN_USE + ERROR status. CH_Dependents() is called to propagate state change. Super component FTP_Dependson_CH() is called.
CH_Dependents():	NI_Dependson_CH() is called with FAIL message.
FTP_Dependson_CH():	Voter status checked. FTP set to RECOVERING because of error on CH(1).
NI_Dependson_CH():	NI[1] set to FAILED + IN_USE + ERROR status (effect of FAIL message from CH). NET1_Dependson_NI() is called with ERROR message.
NET1_Dependson_NI():	NI[1] is controller (IN_USE) and sends ERROR, so NET1 sets RECOVERING. NET1 also sets all children (NODES and LINKS) to GOOD + NOT_IN_USE. NODE_Dependents() and LINK_Dependents() functions are called with NOT_IN_USE message. (LINK_Dependents are not traced from this point.)
NODE_Dependents():	For each NODE, message from parent NET is interrogated and corresponding effect message is sent to the node's attached DIU (if one exists). In this case, four nodes send a NOT_IN_USE message to their DIU's.
DIU_Dependson_NODE():	In response to the NODE message, the DIU sets its mode to NOT_IN_USE. Device component function, DEVn_Dependson_DIU(), is called with NOT_IN_USE message.
DEVn_Dependson_DIU():	In response to the NOT_IN_USE message sent from the DIU, the I/O device sets its mode to NOT_IN_USE also. Because the I/O devices are quad redundant, super component DEVICE_Dependson_DEVn() is called.
DEVICE_Dependson_DEVn():	Voter status checked. DEVICE is not set to RECOVER because device error is not present (being NOT_IN_USE is not an error condition).

Upon return to ASSIST, the following state exists:

CH[1] is set to ERROR. NI[1] is set to ERROR.

FTP is set to RECOVERING. NET1 is set to RECOVERING.

All NODES, LINKS, DIU's, and DEVICES on NET1 are set to NOT_IN_USE.

A component such as the DEVn can be both GOOD and NOT_IN_USE and still fail to a state of FAILED, ERROR, and NOT_IN_USE. If this failure occurs upon restoration of the network when the DEVn status is changed from NOT_IN_USE to IN_USE, then the super component function DEVICE_Dependson_DEVn detects the error and sets a recovery for the DEVICE.

Results. Because the FMES is an extension of ASSIST, results are only available for serial ASSURE and parallel ASSURE. When run on a SUN 3/150 processor, serial ASSURE took 6.2 hours and produced over 27 million transitions. If states are not aggregated, then the number of transitions is equivalent to the number of states. Many of the states are equivalent, and direct comparison with other tools that would aggregate these states is difficult. However, that this system of dynamically reconfiguring mesh networks was analyzed in reasonable time on an ordinary computer is an accomplishment that has not been achieved before. Parallel ASSURE (again using a 32-node hypercube) solved this model in a scant 1.3 minutes. It is expected that large fault-tolerant systems, typical of those found in today's avionic architectures, can now be analyzed using FMES and Parallel ASSURE.

Concluding Remarks

The reliability model generator (RMG) and ASSURE are prototype programs that have been developed to test advanced concepts for the reliability analysis of future fault-tolerant flight control systems. Results of tests using RMG indicate that the automated failure modes-effects analysis (FMEA) algorithm embedded in RMG successfully generated an accurate reliability model from a graphical block diagram of the system. A drawback of this technique may be that the model size is almost

three times larger for the automatically generated graphical model.

Combining the processes of the ASSIST and SURE programs, the ASSURE program eliminates the need to produce and maintain the complete model state space in memory. Solving a model with over 40 000 states and 1 000 000 transitions, the ASSURE program execution was 10 times faster than ASSIST/SURE and used 100 times less memory. Another feature of ASSURE is that its solution technique can be parallelized and thus can be executed on parallel computers such as the hypercube. When this same model was run on a 32-node hypercube, another hundredfold increase in performance over serial ASSURE was obtained.

To better model complex redundancy management processes, the ASSIST language syntax was extended in ASSURE to allow function calls to C language procedures. Drawing on the automated FMEA approach pioneered with RMG, a modelling technique called failure modes-effects simulation was used to model a large system consisting of one quad fault-tolerant computer, two mesh networks, and several quad redundant input/output devices. The system contained over 80 components and 7 redundancy management groups overall. This system produced over 27 million transitions and took 6.5 hours to complete using the serial version of ASSURE. The parallel version was completed in 1.3 minutes.

These results indicate that the techniques are available to represent and solve large, complex reliability models of integrated and distributed flight control systems.

NASA Langley Research Center
Hampton, VA 23681-0001
July 23, 1992

References

1. Cohen, G. C.; Lee, C. W.; Strickland, M. J.; and Torkelson, T. C.: *Final Report: Design of an Integrated Airframe/Propulsion Control System Architecture.* NASA CR-182007, 1990.

2. Bavuso, Salvatore J.; Dugan, Joanne Bechta; Trivedi, Kishor; Rothmann, Beth; and Boyd, Mark: *Applications of the Hybrid Automated Reliability Predictor.* NASA TP-2760, 1987.

3. White, Allan L.: *Upper and Lower Bounds for Semi-Markov Reliability Models of Reconfigurable Systems.* NASA CR-172340, 1984.

4. Butler, Ricky W.; and White, Allan L.: *SURE Reliability Analysis—Program and Mathematics.* NASA TP-2764, 1988.

5. Butler, Ricky W.: An Abstract Language for Specifying Markov Reliability Models. *IEEE Trans. Reliab.*, vol. R-35, no. 5, Dec. 1986, pp. 595–601.

6. Johnson, Sally C.: *ASSIST User's Manual.* NASA TM-87735, 1986.

7. Cohen, G. C.; and McCann, C. M.: *Reliability Model Generator Specification.* NASA CR-182005, 1990.

8. McCann, Catherine M.; and Palumbo, Daniel L.: Reliability Model Generator for Fault-Tolerant Systems. AIAA-88-4435, Sept. 1988.

9. Lala, Jaynarayan H.; Harper, Richard E.; Jaskowiak, Kenneth R.; Rosch, Gene; Alger, Linda S.; and Schor, Andrei L.: Advanced Information Processing System (AIPS)-Based Fault Tolerant Avionics Architecture for Launch Vehicles. *Proceedings IEEE/AIAA/NASA 9th Digital Avionics Systems Conference*, IEEE Catalog No. 90CH2929-8, Inst. of Electrical and Electronics Engineers, Inc., 1990, pp. 125–132.

10. Palumbo, Daniel L.: Three Approaches to Reliability Analysis. *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference—NAECON 1989*, Volume 1, IEEE Catalog No. 89CH2759-9, Inst. of Electrical and Electronics Engineers, Inc., 1989, pp. 308–315.